Assignment 3 – Software Design and Modelling

Refactoring the Design of an Existing Project

Claudio Maggioni Raffaele Morganti

1 Project Selection

The aim of this project is to refactor a part of or a complete application, to achieve good software design without changing the application behaviour. No restrictions are placed on the size or type of project, other than being hosted on GitHub as a public repository.

We choose the project **dtschust/Zork**, which is inspired by the namesake game *Zork*[1] released in the early 1980s.

This project is written in Java and it provides a configurable framework of text-based adventure game mechanics. The program is able to parse and execute a given story, which must be provided in an XML file following a specific format.

We choose this project mainly for two reasons:

- It is small enough to be deeply understood and refactored in a couple of weeks. According to *cloc* and as shown in fig. 1, the project has less than 2000 lines of executable code;
- The project has a large number of design anti-patterns, and we think we have an opportunity to perform a significant and interesting refactor.

Language	Files	Blank	Comment	Code
Java	1	114	77	1264
XML	10	0	0	530
Text	1	1	0	83
make	1	1	0	4
Markdown	1	1	0	3
Properties	1	0	2	3
Total	15	117	79	1887

Figure 1: Output of the *cloc* tool for the **dtschust/Zork** project before refactoring.

2 The Project Before Refactoring

A copy of the contents of **dtschust/Zork** can be found in branch `main` of the repository

usi-si-teaching/msde/2022-2023/software-design-and-modeling/assignment-3-refactoring/group7

on *gitlab.com*.

As figure 1 proves, all the implementation code is contained in a single Java file. A total of 1264 code lines, 77 comments, and 114 blank lines are found. As the project is composed by only 11 classes, we decide to manually inspect the source code to find instances of bad design that we can improve with refactoring.

[1] `https://en.wikipedia.org/wiki/Zork`

The *Zork* class is over 1200 lines long (including blanks and comments) and handles almost all the application logic, making it an obvious instance of the god class anti-pattern. The class violates the single choice responsibility principle, as it handles both XML parsing and the actual execution of the given story. Given that the XML story file specification is non-trivial, the parsing logic may be modular. However, the relevant code is all placed in a single method, as shown in listing 1.

```
Element rootElement = doc.getDocumentElement();

/* Every single first generation child is a room, container, creature, or item.  So
    load them in*/
NodeList nodes = rootElement.getChildNodes();
for (k=0;k<nodes.getLength();k++)
{
    Node node = nodes.item(k);
    Element element;
    if (node instanceof Element)
    {
    /* [511 lines omitted] */
    }
}
```

Listing 1: An excerpt from the XML parsing logic found in the *Zork* class. All the contents of the files are parsed in this section without any delegation to other methods or classes other than the Java DOM API.

5 of the other 10 classes, namely *ZorkObject*, *ZorkCreature*, *ZorkItem*, *ZorkContainer* and *ZorkRoom* do not contain any methods, making them data clumps. The sources for *ZorkContainer* are shown in listing 2 as an example of this. Attributes in classes are generally public and mutable, allowing other classes to modify their internal state. This indicates low encapsulation and tight coupling between the modules in this application.

```
class ZorkContainer extends ZorkObject
{
  public String name;
  public HashMap<String,String> item = new HashMap<String,String>();
  public String description;
  public ArrayList<String> accept = new ArrayList<String>();
  boolean isOpen;
  public ZorkContainer()
  {
  }
}
```

Listing 2: Complete source code for the *ZorkContainer* class before refactoring.

The class *ZorkTrigger* encodes an action to be performed in the game state when a given condition is met. The class however, is subject to feature envy by the aforementioned *Zork* class. While condition evaluation is correctly implemented inside it (specifically in the method *boolean evaluate(Zork)*), the action executing code is instead placed in the god class. Moreover, the condition evaluation is itself envious of *Zork* as it takes the responsibility to fetch the current user input (stored in field *String Zork.userInput*). Listing 3 shows the code where this takes place.

```
class ZorkCommand extends ZorkCondition
{
  String command;
  public boolean evaluate(Zork zork)
  {
    if (command.equals(zork.userInput))
      return true;
    else
```

```
9           return false;
10      }
11  }
```

Listing 3: The class *ZorkCommand* (a sub-class of *ZorkCondition*) is envious of how to fetch the current user input by retrieving it from a public field in *Zork* class.

Finally, we execute the *Sonarqube* tool to have a wider overview of the project in terms of metrics. A summary of the *Sonar Scanner* output is shown in fig. 2. Unsurprisingly the results are not stellar. In particular, we want to point out the "Cognitive Complexity" metric, which is related to the number of boolean conditions checked in methods. Our refactoring manages to lower its value by a factor of 6, showing how the project originally has a very high density of logic per method.

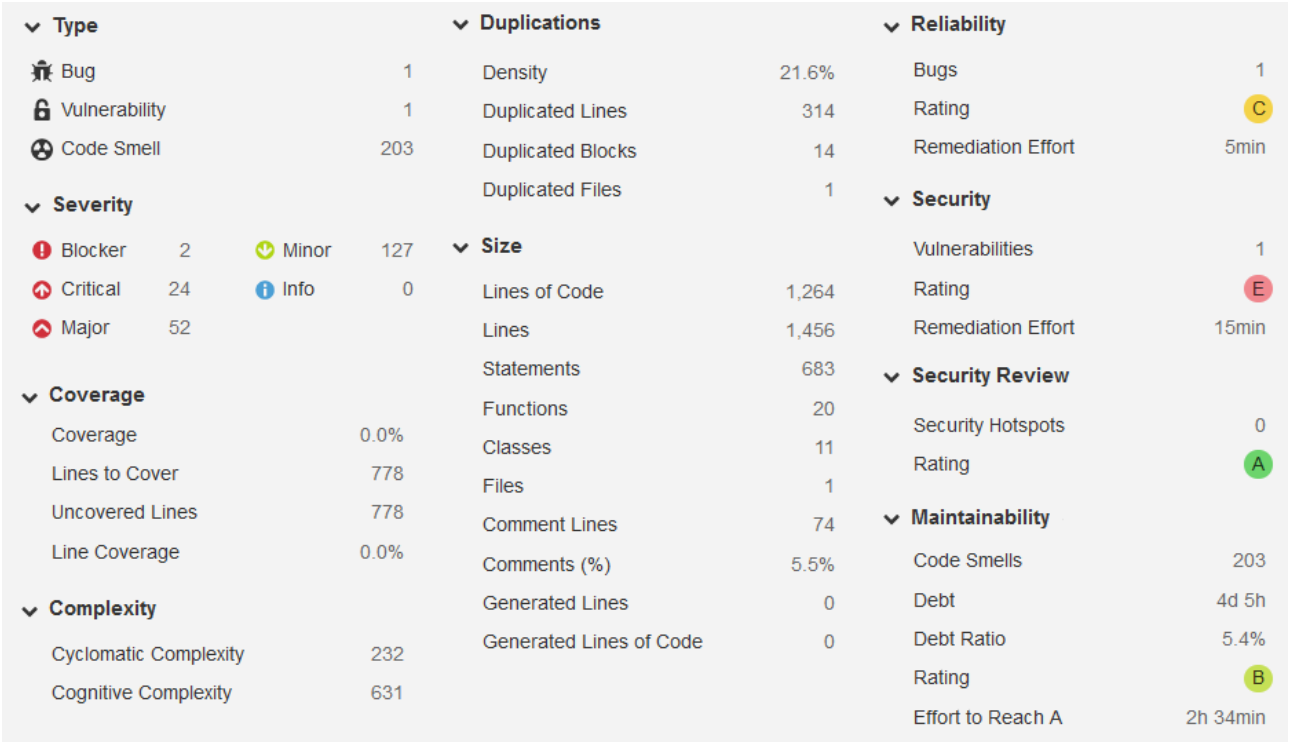

Figure 2: Summary of the *Sonar Scanner* detection on the initial project

In conclusion, we determine that the code in **dtschust/Zork** does not follow OOP design best practices, in particular falling short of providing loose coupling and defining an effective separation of concerns between its classes. Our refactor aims to introduce better design to the project to increase modularity, which in turn increases the testability of each class and the readability of the overall code base.

3 Refactoring

A copy our refactor of **dtschust/Zork** can be found in branch `refactored` of the repository

usi-si-teaching/msde/2022-2023/software-design-and-modeling/assignment-3-refactoring/group7

on *gitlab.com*.

3.1 Source Code Structure and Build System

The first change we make on the repository is to introduce *Maven* as a build system and following its standard directory structure mandated for source code and tests. We also split the *Zork.java* file in several files, one per class, and we move each class under the *com.github.dtschust.zork* package as placing classes in the default package (their original position) is considered bad practice in Java.

These steps are relatively trivial but however crucial to make our work easier with further refactoring efforts. Additionally, this simplified the implementation of (originally missing) test code to check the behaviour of the program remains the same.

3.2 XML Parsing Logic

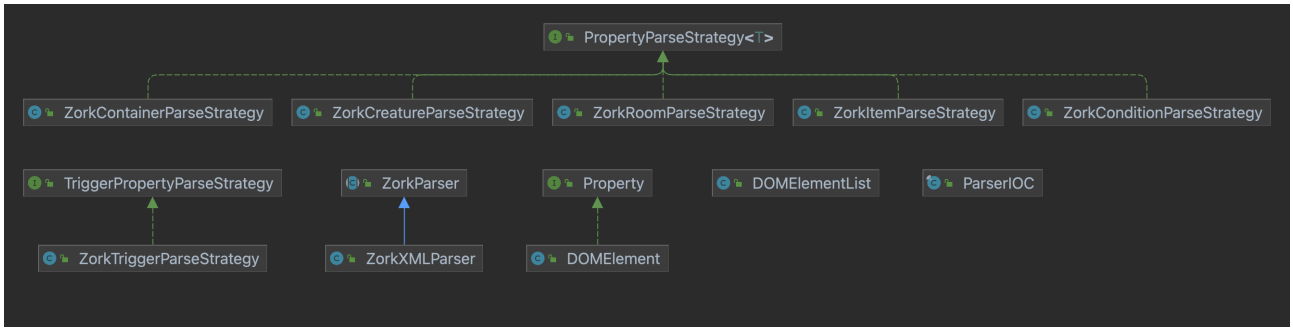

Figure 3: UML class diagram of the *com.github.dtschust.zork.parser* package

We then separate the XML parsing logic in the package *com.github.dtschust.zork.parser*, whose UML class diagram is shown in fig. 3. Here, the XML-specific logic is separate from the application-specific allocation of the story data structure.

DOM elements and lists of elements are encapsulated in the *dom.DOMElement* and *dom.DOMElementList* proxy classes. These classes in turn are used to implement an encoding-agnostic interface named *Property*, which abstracts the story file to a collection of property names, property values and sub-properties.

At each nesting level a different parsing strategy is defined as a class under the *strategy* package. Each class implements a method that takes a *Property* and returns an instance of the matching application-specific entity. Each strategy may depend to other strategies through dependency injection, with inversion of control handled by the *ParserIOC* class.

The parser is now a collection of 11 classes and 3 interfaces with a single entry point. With the line of code `ParserIOC.xmlParser().parse(filename, System.out)` the entire story data structure is parsed from the XML file and instantiated. Our new design increases testability by separating each nesting level into easily mockable classes. Moreover, the new abstractions allow for easy implementation of parsers for other encodings.

3.3 User Actions

```
1 public class ReadAction implements Action {
2     @Override
3     public boolean matchesInput(List<String> arguments) {
4         return arguments.get(0).equals("read");
5     }
6
7     @Override
8     public int getMinimumArgCount() {
```

```
        return 2;
    }

    @Override
    public boolean run(ZorkGame game, List<String> arguments) {
        return game.getItem(arguments.get(1)).map(i -> {
            game.stream.println(i.getWriting());
            return true;
        }).orElse(false);
    }
}
```

Listing 4: Implementation of the *read* action in the command class *action.ReadAction*.

We decide to implement the execution of user invoked or triggered actions as a variation of the command pattern. All this logic is now in the package *com.github.dtschust.zork.repl*.

All actions are implemented as a command class under the *action* package, which implement a method `run` taking as parameters the story data and returning a *boolean* indicating success of failure. An example of command class is shown in listing 4.

The dispatcher is implemented in the class *ActionDispatcher*. Our variation of the command pattern makes this dispatcher responsible for parsing the action string coming from triggers and the user, choosing the right action to invoke aided by methods in each command class, and dispatching the command providing the parsed action arguments as parameters of the method *run*. This variation is done to cater the domain-specific need of providing a text based REPL (Read-eval-print loop) like interface to the user.

3.4 Story Data

In the overall project we also perform several refactor on the classes modelling the story data and the game state. Fields on these classes are now immutable whenever possible and always *private* to provide better information hiding and to loosen coupling. Additionally, the feature envy anti-patterns are solved by moving the respective methods into the classes they fit in. This required an extensive refactor of the the trigger-related logic from methods in class *Zork* to methods in *GameData* and improved condition evaluation methods in the *ZorkCondition* hierarchy exploiting dynamic dispatch.

4 Testing

A problem we found was the absence of tests, so we needed to write them to ensure to don't introduce behavioral changes.

The original GitHub repository includes the two files named *sampleGame.xml* and *RunThroughResults.txt*. The first contains the game configuration (with the definition of all the rooms, items, creatures, etc. in the game). The latter is instead a log-file containing a sequence of inputs and their respective output.

We used these files to build a system test. In particular, our test is starting the game in a thread and mocks the default system input/output interface to automatically send the commands to the game. The implementation logic of this test is shown in listing 5.

```
@Test
void testSampleGame() {
    String gameConfig = "sampleGame.xml";
    String gameExecution = "RunThroughResults.txt";

    CommandReader run = new CommandReader(gameExecution);
```

```
7       IOWrapper io = new IOWrapper(true);
8       new Thread(() -> {
9           try {
10              catchSystemExit(() -> Zork.runZork(gameConfig));
11          } catch (Exception ignored) {}
12      }).start();
13      while(true){
14          switch(run.getInstructionType()) {
15              case SEND:
16                  io.write(run.getInstruction());
17                  break;
18              case RECV:
19                  assertEquals(run.getInstruction(), io.read());
20                  break;
21              default:
22                  io.restore();
23                  return;
24          }
25      }
26  }
```

Listing 5: Code of the system test we implemented.

Although it doesn't cover all of the possible outcomes, it was the best option available to ensure the behavioral consistency. Since this game has been realized as an assignment for a university course, a *zorkRequirements.pdf* file with all the specification is available, and we also relied on it.

To verify the behavior of the initial project, we don't added tests to the original sources, but to the ones already converted to a Maven project for convenience.

A copy of the tests added to the **dtschust/Zork** can be found in branch `main-with-test` of the repository

usi-si-teaching/msde/2022-2023/software-design-and-modeling/assignment-3-refactoring/group7

on *gitlab.com*.

5 Conclusions

At the end of the refactoring we ran again *Sonar Scanner*, with a summary of the results in fig. 4.

By comparing these with the ones in fig. 4 described at the start of this report, we can asses there is a huge improvement. The main benefits are the reduced "cognitive complexity" from a score of 631 to 108 and the removal of code duplication.

In the end the project size is almost identical, however the source deeply changed, with the initial 20 methods and 11 classes extracted in a total of 171 methods and 47 classes. Now the code takes advantage from the basic object-oriented programming principles.

At the first impact the code wasn't really easy to understand, so our first steps in our refactor process were the method extraction of logically separated tasks. After that we started to extract classes and find some design-patterns (like the command pattern described above). Only in the end we cared about the visibility of the attributes and we enforced encapsulation. By working with this step-by-step strategy made our job easier and we didn't encountered any insurmountable barrier. Approaching to refactor in a continuous way allowed us to don't stuck with too hard and too big changes to be done in a single run. If we need to state what was the hardest task, we found the implementation of the command pattern the most challenging overall.

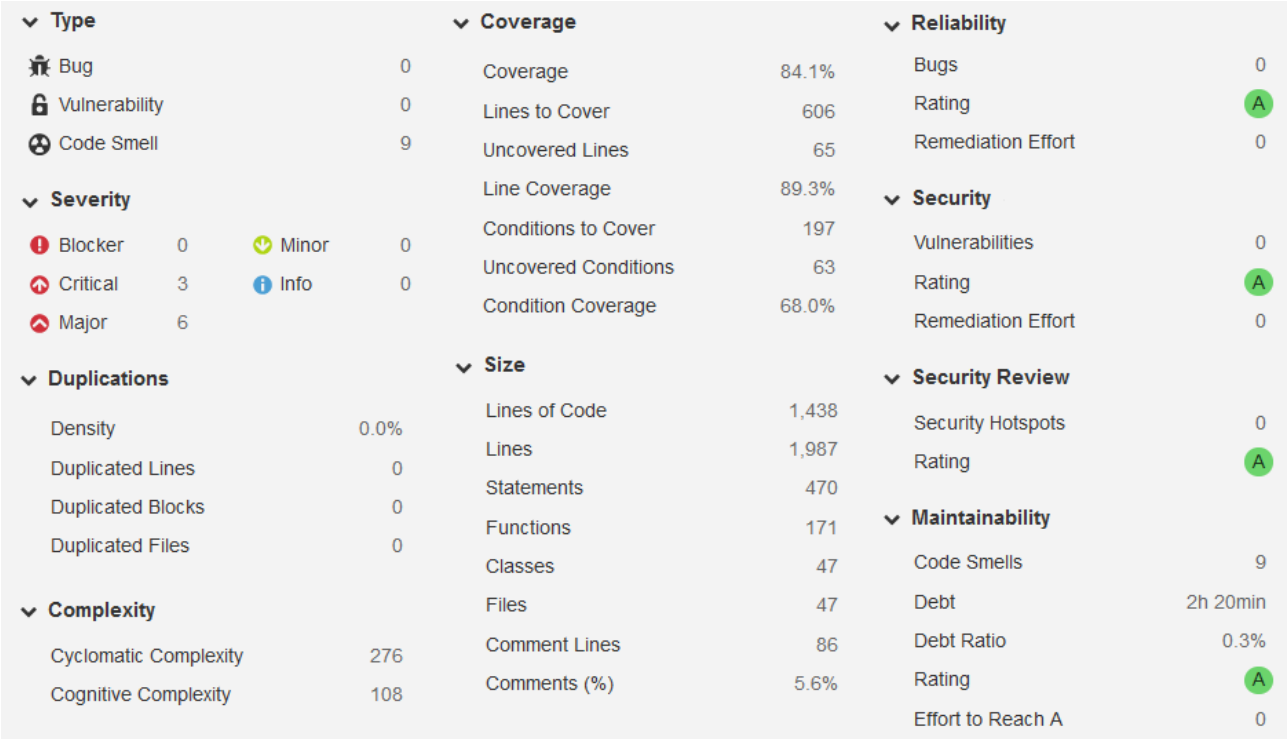

Figure 4: Summary of the *Sonar Scanner* detection on the refactored project

6 Future work

As always happens when talking about refactoring, it will be always possible to do something more. However we think our extensive refactor improved all aspects of this project and removed all the anti-patterns.

Still some improvements are possible, as also the last run of *Sonar Scanner* shows in the "code smells" section. In particular the 3 detected as "critical" are related to the usage of the generics (listing 6) and they can be removed. However since in all the three cases these generics are bounded we don't see to this as a high priority issue and we consider the required effort bigger than any potential benefit of fixing these.

```
1 // class ZorkGame
2 public Optional<? extends ZorkObject> getObject(final String objectName);
3 //class Property
4 List<? extends Property> subProperties();
5 List<? extends Property> subPropertiesByName(String name);
```

Listing 6: Usage of wildcard types in the code.

The 6 "major" issues instead, are less interesting:

- Two of them are about the number of parameters in the constructor for the *ZorkRoom* and *ZorkCreature* classes, but they are instantiated through the respective *StrategyParser.parser* method that is working as a builder.
- The other four are related to the usage of calls to the *System.out* (all of them are sort of logs printed when some exceptions are thrown if trying to load of a wrong game configuration file). Making more informative logs of the causes of these exception would certainly be useful, but this improvement is out of the scope of refactoring.

In conclusion, although *Sonar Scanner* detected them, we don't think they are really relevant.